Lecture 25 — Suffix Arrays

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Kanat Tangwongsan — April 17, 2012

Material in this lecture: The main theme of this lecture is *suffix arrays*.

- Definition: Suffix arrays
- Sorting all the suffixes faster than $O(n^2)$.

Here is a standard string-processing problem: find a pattern P in a big corpus of text S. For context, consider the following:

> Suppose we're given a DNA sequence S (e.g., $S =$ `AGTCG.....`), and we're interested preprocessing the string S so that after that, we can locate all occurrences of a pattern P ($|P| \ll |S|$) in S fast (say $O(|P| \log |S|)$ or $O(|P| + \log |S|)$)[1]

Last week, we discussed dynamic programming, where the main idea was to identify redundant computation and reuse the results whenever possible. Today, we are going to see another application of this idea, disguised in the context of string algorithms.

1 Suffix Arrays

From searching for a string pattern to finding the longest repeated substrings to computing the maximal palindromes, suffix arrays and its close cousin got it covered. Unsurprisingly, they are heavily used in bioinformatics applications (finding patterns in protein/DNA) and in data compression (Burrows-Wheeler transform; see `bzip2`) just to give a few examples.

A suffix array is a data structure for efficiently answering queries on large strings. The *suffix array* of a string S is an array giving pointers to the suffixes of S sorted in lexicographical order of the suffixes. For concreteness, let $\texttt{suff}_i(S)$ denote the suffix of S starting at position i.

As an example, consider the string $S =$ `parallel`. The suffixes of S are:

i	$\texttt{suff}_i(S)$
0	`parallel`
1	`arallel`
2	`rallel`
3	`allel`
4	`llel`
5	`lel`
6	`el`
7	`l`

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1] This is actually better per-pattern than KMP or similar string search algorithms, which require about $O(|P| + |S|)$.

Sorting these suffixes, we get

```
allel
arallel
el
l
lel
llel
parallel
rallel
```

which results in the following suffix array:

$i =$	0	1	2	3	4	5	6	7
SA[i]	3	1	6	7	5	4	0	2

Notice that in this example, the suffix array indicates that $\texttt{suff}_3 < \texttt{suff}_1 < \texttt{suff}_6 < \texttt{suff}_7 < \texttt{suff}_5 < \ldots$, where $<$ denotes the lexicographical ordering.

1.1 String Matching Using Suffix Array

Let's go back to the problem posed at the beginning of the lecture. We'll pretend for a second that we know how to construct suffix arrays. What can we do now?

Remember that we wanted to find occurrences of a pattern P in S. It is not hard to see that the pattern P occurs at position i of S if and only if P is a prefix of the suffix $\texttt{suff}_i(S)$. This means

> the occurrences of P in S = all suffixes of S having P as a prefix.

This observation allows us to transform the string matching problem into a prefix matching problem on all the suffixes.

But we haven't actually made any progress. To proceed, we will make use of the sortedness property of suffix arrays. Since suffix arrays are sorted, we could perform a binary search to locate an occurrence of the pattern. The number of comparisons required by a binary search is logarithmic in the size of the domain (i.e., the length of the suffix array, which has $|S|$ entries). Each comparison is made between P and a suffix of S, which we know can be done in $O(|P|)$ work. Therefore, for a total of $O(|P| \log |S|)$ work, we can locate an occurrence of P, or report that none exist.

Finding Them All. If our goal is to find all occurrences of P, we'll need an additional an observation. The following observation follows from the sortedness of suffix arrays:

> **Observation:** For any prefix A, all suffixes of S that have prefix A are contiguous entries in the suffix array.

Therefore, if we're looking for the pattern $P = \texttt{l}$ in the string $S = \texttt{parallel}$, according to these observations, we should be looking for suffixes that begin with `l` and we know that all the

occurrences are next to each other in the suffix array of S. Indeed, for the string $S = \texttt{parallel}$, the suffixes `l`, `lel`, and `llel`—all the suffixes that begin with $P = \texttt{l}$—are next to each other.

Given this setup, our task now boils down to locating the starting position. This can be easily done using a binary search, which requires $O(\log |S|)$ string comparisons since the suffix array of S contains $|S|$ entries. As each comparsion can be performed in $O(|P|)$ work, the total work is $O(|P| \log |S|)$.

Remarks: With a bit more work, this can be improved to $O(|P| + \log |S|)$. For comparsion, most efficient string searching algorithms (e.g., Knuth-Morris-Pratt and Rabin-Karp) allow us to find occurrences of a pattern P in a string S in time $O(|S| + |P|)$, so preprocessing the string S can be a big win if we're performing multiple searches.

1.2 How to Build A Suffix Array?

We have just seen an application of suffix arrays. How can we build one given an input string?

As the definition suggests, the most straightforward way to build a suffix array is to enumerate all the suffixes of a given string and sort them. On an input string of length n, the algorithm will perform $O(n \log n)$ comparisons (if an efficient comparision-based sorting algorithm is used), but unfortunately, each comparison is rather costly: comparing two strings of length n costs $O(n)$ work. Therefore, the overall work of this simple approach is $O(n^2 \log n)$. This is slow for large input.

Exercise 1. *Given two strings S and T of length n, what's the work and span of comparing them? What does this say about the span of the naïve algorithm for suffix arrays?*

2 Building A Suffix Array in Less Than $O(n^2)$ Work

One thing is clear about the above algorithm: we're doing a lot of redundant work comparing strings. While sorting n arbitrary length-n strings in a comparison model will need $\Omega(n^2 \log n)$ work, the strings we're sorting aren't arbitrarily given. After all, we're working with the suffixes of a string. In this part of the lecture, we'll focus on improving the work done in comparing strings.

The goal is to be able to build a suffix array in $O(n \log^2 n)$ work. The current state-of-the-art is $O(n)$ (which is also parallel), but this is beyond the scope of the class.

Say we want to compare two strings $X = x_0 x_1 \ldots x_{n-1}$ and $Y = y_0 y_1 \ldots y_{m-1}$, which have lengths n and m. The simplest implementation costs $O(\min(n, m))$ work as it would compare x_0 against y_0 and returns an answer if x_0 differs from y_0, but otherwise, it proceeds to compare x_1 against y_1 so on so forth until it finds a location where the two strings differ. In code, we have:

```
fun strcmp (X, Y) =
  case (X, Y)
    of (nil, nil) => EQUAL
     | (nil, _) => LESS
     | (_, nil) => GREATER
     | (x::X',y::Y') =>
        (case compare(x, y)
           of EQUAL => strcmp(X', Y')
```

```
                | r => r)
```

For a moment, we'll assume that X and Y have the same length (i.e., $n = m$). We'll also pretend that we have done the comparison $\texttt{suff}_1(X)$ and $\texttt{suff}_1(Y)$ before. With these assumptions, if we have access to the outcome of $\texttt{r = strcmp(suff}_1(X)\texttt{,suff}_1(Y)\texttt{)}$, then we can compare X with Y in $O(1)$: if $x_0 = y_0$, then return r else return $\texttt{compare}(x_0, y_0)$. But to take advantage of this idea, we'll probably need to compute a lot of states, most likely $O(n^2)$ of them. We haven't made any saving.

The idea is to generalize this approach further. We'll experiment with divide and conquer. For simplicity, we'll still assume that $n = m$. In addition, we'll assume that $n = m = 2^k$ is a power of two. Then, we can write X and Y as

$$X = \boxed{X_0}\boxed{X_1}$$
$$Y = \boxed{Y_0}\boxed{Y_1},$$

where $X_0 = x_0 x_1 \ldots x_{n/2}$, $X_1 = x_{n/2+1} \ldots x_{n-1}$, and Y_0 and Y_1 are defined similarly for Y. We have:

Observation: $\texttt{strcmp(X,Y)} = \texttt{case strcmp}(X_0, Y_0)\ \texttt{of EQUAL=>strcmp}(X_1, Y_1)\ \texttt{| r=>r}$

This leads to the following pseudocode implementation, which seems a bit convoluted and doesn't yet help us in any way:

```
fun strcmp (X as X_0X_1, Y as Y_0Y_1) =
  case (X, Y)
    of ... (* handle base cases *)
     | _ =>
       (case strcmp(X_0,Y_0)
          of EQUAL => strcmp(X_1, Y_1)
           | r => r)
```

How might we reuse some of the results to speed up the construction of suffix arrays? Let's try the following simple idea: we'll split every suffix in half and hope that these substrings have been compared before. On our running example of $S = \texttt{parallel}$, we would have the following (we use the special symbol \$, which is lexicographically smaller than any real symbol, to preserve the length):

para	llel
aral	lel$
rall	el$$
alle	l$$$
llel	$$$$
lel$	$$$$
el$$	$$$$
l$$$	$$$$

We quickly realize that if we are to do less work than $O(n^2)$, we couldn't have compared all combinations of these length-$(n/2)$ pairs before. In fact, even if we could, we wouldn't have the space or time to store the information. There are too many pairs after all.

Here is where we need a new idea: instead of storing the outcome of a comparison, we'll "rank" these substrings in a way their ranks can be compared in place of comparing the actual strings. Let $\texttt{rank}_i(S,\ell)$ denote the *rank* of the length-ℓ substring of S starting at i (i.e., the substring $S[i..i+\ell-1]$). The guarantee we make is the following: for any length $\ell > 0$,

$$\texttt{Int.compare}(\texttt{rank}_i(S,\ell),\ \texttt{rank}_j(S,\ell)) = \texttt{String.compare}(S[i..i+\ell-1],\ S[j..j+\ell-1]) \quad (1)$$

One possibility is to rank the smallest string 0, the second smallest string 1, etc. Using this idea, we would assign the ranking:

para: 6	llel: 5
aral: 1	lel$: 4
rall: 7	el$$: 2
alle: 0	l$$$: 3
llel: 5	$$$$: -1
lel$: 4	$$$$: -1
el$$: 2	$$$$: -1
l$$$: 3	$$$$: -1

Sorting The Suffixes of S Using Rank. Before we talk about computing the rank function, let's see how we can use the rank information to sort all the suffixes of a string S. Say we have the rank information for all length $|S|/2$ substrings of S, like in the example above. We can easily sort the suffixes of S using the following 2-step process:

1. Make a sequence $(i, (\texttt{rank}_i(S,|S|/2), \texttt{rank}_{i+n/2}(S,|S|/2)))$.
2. Sort this sequence by the rank pairs.

2.1 How to Compute the Rank Function?

The idea is to construct the rank function recursively. To pursue this idea, there are two questions that need to be answered:

1. *Base Case.* How can we efficiently compute the rank function for $\ell = 1$?
2. *Inductive Case.* How can we quickly derive the rank function for $\ell > 1$ given the rank function for $\ell' < \ell$? Presumably, we want an even split (e.g., $\ell' = \ell/2$) to avoid $O(n)$-depth recursion.

We'll start with the base case. More concretely, the task here is: given a length-n sequence of characters (i.e., the input has type `char seq`), construct a sequence R of length n such that $R[i]$ is an integer denoting the rank of the character at position i of the input.

The simplest approach (as was suggested by the class) is to use the ASCII value of these characters. It is easy to check that this satisfies the rank requirement, see Equation (1). In code, we can write the following:

```
fun rankBase (S : char seq) : int seq =
   tabulate (fn i => Char.ord (nth S i)) (length S)
```

We'll consider a slightly more complicated approach, which hints at what we'll need to do in the recursive case. This alternative also has the advantage that the rank numbers are between 0 and $|S|-1$ and that it works with "black-box" characters (you can only compare elements but you don't have any side information about their ordering).

The following code captures the essence of the rank function in the base case:

```
1  % compute the rank where the i-th element is the rank of S[i..i+ℓ−1]
2  fun rankBase(S) : int seq =
3  let
4     val pairs = ⟨(i, S[i]) : i = 0, ..., |S|−1⟩
5     val sortedPairs = sort (Char.compare o second) pairs
6     val newRank = ⟨(#1 sortPairs[i], i) : i = 0, ..., |S|−1⟩
7  in inject newRank ⟨−1 : i = 0, ..., |S|−1⟩ end
```

It first sorts all the characters and use the index in the sorted sequence as the rank. The sequence *newRank* stores ordered pairs (position, rank), so we can use `inject` to derive the desired sequence.

But as the following example shows, the algorithm doesn't correctly handle repeated characters.

```
  p a r a l l e l

=> (0,p), (1,a), (2,r), (3,a), (4,l), (5,l), (6,e), (7,l)

sort

=> (1,a), (3,a), (6,e), (4,l), (5,l), (7,l), (0,p), (2,r)

assign ranking: (index, rank)
=? (1,0), (3,1), (6,2), (4,3), (5,4), (7,5), (0,6), (2,7)

actually, we want:

== (1,0), (3,0), (6,2), (4,3), (5,3), (7,3), (0,6), (2,7)
```

What we need to do is to make sure that if s and t are the same character, they get the same rank number. Since the sequence is already sorted, this is easily done with a scan (see the lectures on scan).

Inductive Case. Building on the ideas we laid out so far, we could represent each length ℓ string as an ordered pair of rank numbers: that is, we will represent the substring $S[i..i+\ell-1]$ as the ordered pair $(\texttt{rank}_i(S,\ell/2), \texttt{rank}_{i+\ell/2}(S,\ell/2))$ (we're still assuming ℓ is a power of two). This conversion allows us to sort these length-ℓ substrings in $O(n \log n)$ work because each comparison is made between a pair of int pairs, which can be done in constant work, and we only need $O(n \log n)$ comparisons using an efficient sorting algorithm. This leads to the following pseudocode:

```
1  % compute the rank where the i-th element is the rank of S[i..i+ℓ-1]
2  fun rank(S, ℓ) : int seq =
3  if ℓ = 1 then rankBase(S) else
4  let
5     val rank' = rank(S, ℓ/2)
6     val fun rnk i = if (i ≥ |S|) then -1 else rank'[i]
7     val pairs = ⟨ (i, (rnk i, rnk(i + ℓ/2))) : i = 0, ..., |S| - 1 ⟩
8     val sortedPairs = sort (intPairCmp o second) pairs
9     val newRank = ⟨ (#1 sortPairs[i], i) : i = 0, ..., |S| - 1 ⟩
10 in inject newRank ⟨ -1 : i = 0, ..., |S| - 1 ⟩ end
```

Like in the base case, the sequence *newRank* contains ordered pairs (position, rank) so that the `inject` function can be used to construct the desired result; however, the same problem that arose with duplicate strings in the base case also shows up here.

Exercise 2. *The code as writen is broken. Fix the computation newRank when the substrings contain duplicates.*

Analysis. It is not difficult to see that within each recursive call to `rank`, we spend $O(n \log n)$ work to sort and other bookkeeping. Additionally, for $\ell > 1$, we make a recursive call with ℓ half the original ℓ. Therefore, we have the following recurrence: $W(\ell) = W(\ell/2) + O(n \log n)$, with $W(1) = O(n \log n)$ (or $O(n)$ if we're using the first construction for the base case). This solves to $W(n) = O(n \log^2 n)$, which is better than $O(n^2)$, as we promised.

Stay tuned for an SML/NJ implementation of this (as well as a C/C++ implementation) in the Git repository.